

DXQ: A Distributed XQuery Scripting Language

Mary Fernández
AT&T Labs Research
mff@research.att.com

Trevor Jim
AT&T Labs Research
trevor@research.att.com

Kristi Morton
Univ. of Texas Austin
kmorton@cs.utexas.edu

Nicola Onose
UCSD
nicola@cs.ucsd.edu

Jérôme Siméon
IBM Watson Research
simeon@us.ibm.com

ABSTRACT

We present DXQ, an extension of XQuery to support the effective and efficient development of distributed XML applications. A DXQ program can invoke remote DXQ programs both synchronously and asynchronously and can dynamically ship DXQ code to execute at remote servers. We illustrate the power of the language with two distributed applications: the resolution algorithm of the Domain Name System (DNS) and the Narada overlay-network protocol. Our implementation permits concurrent evaluation of DXQ expressions at each server and can produce results extensionally (as XML values) or intensionally (as DXQ expressions).

1. INTRODUCTION

Our goal is to support the effective and efficient development of distributed applications based on XML. We present DXQ, an extension of XQuery with distributed programming features. The declarative nature of XQuery, combined with its support for XML processing, makes it well suited for rapid development of complex distributed systems, in particular Web services, peer-to-peer applications, and distributed resource-management (DRM) applications¹.

We have been experimenting with DXQ on several applications, including the resolution algorithm of the Domain Name System (DNS) and the Narada overlay-network protocol [4], which are representative of more complex DRM applications [9, 17] and of other overlay networks. Both applications raise significant challenges in terms of distribution and data processing.

The Domain Name System (DNS) is a prototypical DRM system. DNS provides the basic infrastructure for resource management in the Internet: name resolution (mapping a hostname to its IP address) and service location (e.g., determining an organization's mail server). DNS was deployed in the early Internet and has grown with it. Today, it comprises a global network of servers, providing over 400 million

¹Not to be confused with the more common acronym for digital rights management.

address records [21], and handling more than 100 million queries per day at over 100 physical root servers [25]. Because of its importance, DNS has been well studied and is a benchmark for the DRM problem. There are many efforts to improve on its performance and features, and remedy its deficiencies [5, 16, 18]. We use DNS as our running example.

Narada is a mesh overlay network in which nodes can dynamically enter and leave the mesh overlay and which provides infrastructure for routing and multicast. Overlay networks like Narada are interesting test cases for DXQ, because they require periodic, asynchronous messages to maintain protocol state, and the state includes membership and routing tables that can be naturally described and manipulated in XML. Space constraints prevent us from discussing the details of our Narada implementation, but more information is available from our Web site.

Implementing DRM protocols is non-trivial. Resource descriptions are typically distributed, because they are co-located with the resources that they describe, and volatile, because they include physical parameters that vary over time. More significantly, distributed participants control their own resources and have their own resource management *policies*. This means no universal policy, or program, can be applied by any one participant to manage a distributed resource. Instead, multiple systems participate in a protocol, each applying their own local policy. Participants and their resources tend to be organized hierarchically, and the set of participants relevant to a particular task may be *dynamic*, that is, dependent on the other participants and the accessed resources.

Most network applications and protocols are stateful, so DXQ includes the update and mutable-variable features of XQueryP [3]. In addition, DXQ supports synchronous and asynchronous invocations of remote DXQ programs, and the ability to ship DXQ code to remote locations. Our implementation permits concurrent evaluation of DXQ expressions at each server and can produce results extensionally (as XML values) or intensionally (as DXQ expressions).

A DXQ server is a *complete* query processor for XML, and can act as a client as well as a server. It exports a module written in DXQ, and accepts arbitrary DXQ queries that can invoke the server's exported functions and even cause the server to query other servers. For example, a DNS resolver works by making a series of database queries, starting at a root name server, and following delegations until reaching a name server that has the final answer (a hostname's IP address). This series of queries is naturally expressed as a single DXQ query, where following a delegation corresponds

to a join of remote databases, computed at the resolver (the client). In contrast, a multicast can be expressed as a DXQ query that causes the server to forward messages to all child servers in the multicast tree, on behalf of the original client.

DXQ is implemented and is publicly available as part of the Galax XQuery processor. A demonstration of DXQ also appears in the SIGMOD 2007 Demonstrations program [6]. More information about the DXQ project can be found on line at <http://db.ucsd.edu/dxq/>.

We introduce DXQ extensions to XQuery in Section 2, using DNS resolution as a motivating example. In Section 3, we give further examples of how DXQ’s features are needed to handle real distributed protocols. Section 4 describes our implementation, and Section 5 presents related work. We conclude in Section 6 by identifying open design and implementation problems and consider opportunities for DXQ in other application domains.

2. THE DXQ LANGUAGE

The DXQ language includes all of XQueryP [3] and extends it with the features summarized in Figure 1.

First, DXQ distinguishes a module’s interface from its implementation. In XQuery, these are inseparable, and the implementation of an imported module is always visible within the importing module. In DXQ, multiple servers can export the *same* module interface, but each provides its own, potentially unique implementation. The syntax for module interfaces is based on a simplified version of the XQuery prolog without setters, with only variable and function signature declarations, and that permits recursive imports of other module interfaces. In our DNS example, each DXQ server exports the following interface:

```
interface namespace Server = "http://www.dns.org/Server";
declare function Server:resources();
declare function Server:multicast($msg);
```

The `resources()` function returns a server’s DNS resource records, and the `multicast()` function sends a multicast message to child nameservers contained in a server’s delegation hierarchy.

Figure 2 contains a module that implements the DNS server interface. The server that exports this module is the “start of authority” nameserver for the `research.att.com` domain and is named `ns.research.att.com`, as represented by the `soa` element. The definition of `resources()` includes the hostname and IP address for hosts in that domain (the `a` elements) as well as the hostname and IP address for the root nameserver known as “.” name server (the `ns` element). (Here, `resources()` returns literal XML values, but, in practice, it might access resource records stored in a database.)

Second, DXQ adds an `import-interface` declaration to the query prolog, which permits a client query to refer to functions in a module exported by a DXQ server. Figure 3 contains a client program that implements the DNS resolver algorithm. It imports the DNS server interface on Line 1, that it, it associates the namespace prefix `Server` with the module interface in `http://www.dns.org/Server`.

Third, DXQ adds a `let-server-implement` expression, which dynamically asserts that a DXQ server at a particular URI implements an imported interface. Line 5 of Figure 3 asserts that the server at the URI defined by `$x` implements the module interface `Server`, and that this implementation can be addressed with the prefix `S`. In particular, the prefix `S`

```
module namespace Server implements "http://www.dns.org/Server";
declare function Server:resources() {
  <rr>
    <soa dom="research.att.com"
      serv="ns.research.att.com"/>
    <ns dom="." serv="a.root-servers.net"/>
    <a host="a.root-servers.net" addr="198.41.0.4"/>
    <a host="www.research.att.com" addr="192.20.3.54"/>
    <!-- Other hosts in research.att.com -->
  </rr>
};
(: Multicast defined in Figure 5 :)
```

Figure 2: Resource records for `research.att.com`

```
1. import interface Server = "http://www.dns.org/Server";
2. import module U = "DNSUtility";
3. declare function Resolver:lookup($x,$n) {
4.   <rr>{
5.     let server S implement Server at $x return
6.     let $rr := from server S return S:resources()
7.     return
8.     $rr/a[@host=$n],
9.     (for $ns in $rr/ns, $a in $rr/a
10.      where $ns/@serv=$a/@host
11.        and fn:not($ns/@dom = ".")
12.        and U:hostname-lt($ns/@dom,$n)
13.        return Resolver:lookup($a/@addr, $n)/a)
14.   }</rr>
15. };
```

Figure 3: Core DNS resolver in DXQ

associates a server location with a module interface and implementation. Using namespace prefixes to refer to servers further overloads the semantics of namespace prefixes, but permits re-use of the XQuery function call syntax to call remote functions.

Lastly, DXQ includes two remote-evaluation expressions, `from-server-return` and `at-server-do`, which evaluate an expression at a DXQ server synchronously and asynchronously, respectively. The first argument of each expression is the namespace prefix of a server bound by an enclosing `let-server-implement` expression. The second argument is an expression that is evaluated by the given DXQ server. A remote expression may contain calls to remote functions, whose implementations are determined by an enclosing `let-server-implement` expression.

The language extension is small, but provides a powerful mechanism for distributed XML programming. For example, the core DNS resolver algorithm in Figure 3 is implemented by the ten-line, recursive `lookup` function. Uses of the DXQ extensions are indicated by the italicized code on Lines 1, 5, and 6. The `lookup` function takes two arguments: a hostname `$n` to resolve, and the server address `$x` at which to initiate resolution. The algorithm contains one `from-server-return` expression (Line 6), which calls the remote function `S:resources`. This call returns all the server’s resource records to the resolver. The resolver computes the result, which includes all address records for host `$n` defined at the server `S` (Line 8), and the address records for `$n` defined by name servers known to server `S` (Lines 9–13). The latter set is computed by calling `lookup` recursively (Line 13) for each name server known to server `S` whose domain includes `$n` (Lines 11–12). The function `U:hostname-lt` call on Line 12 checks lexicographically that the hostname `$n` is contained in a domain name, forcing the recursion down the delegation of nameservers. This imple-

```

Interface ::= interface namespace NCName = URILiteral ; InterfaceProlog
Module ::= module namespace NCName = URILiteral (implements URILiteral)?; ModuleProlog
InterfaceImport ::= import interface namespace NCName = URILiteral
Expr ::= ...
      | let server NCName implement NCName at Expr return Expr
      | from server NCName return Expr
      | at server NCName do Expr

```

Figure 1: DXQ grammar extensions

```

1. declare variable $cache := <cache/>;
2. declare updating function Resolver:lookup($x,$n) {
3. <rr>{
4. let $ac := $cache/entry[a/@host=$n] return
5. if (empty($ac)) then {
6. let server S implement Server at $x return
7. let $rr := S:resources()
8. let $ans := $rr/a[@host=$n]
9. return {
10. if (empty($ans)) then ()
11. else
12. do insert
13. <entry serv="$x">{$ans}</entry>
14. into $cache;
15. ($ans,
16. ... Lines 9-13 in Figure 2...) }
17. else $ac/a
18. }</rr>
19. };

```

Figure 4: A caching resolver, using updates

mentation returns *all* address resolutions, although another formulation might return only the first match found.

Calling one remote function synchronously, as on Line 6, is a common idiom and is abbreviated by the function call itself. For example, in Figure 3, the following line:

```
let $rr := from server S return S:resources()
```

can be replaced by `let $rr := S:resources()`. The remaining examples use this abbreviated form.

3. USING DXQ IN PRACTICE

The DNS resolver in Figure 3 is implemented almost completely in XQuery proper and requires only modest extensions to express the distributed computation. Implementing more complex protocols, like the Narada overlay network, requires periodic, asynchronous messages and updates to maintain protocol state. The remaining examples illustrate how those features can be implemented in DXQ.

3.1 Maintaining state

Most DRM protocols must maintain and update state, e.g., caches, sequence numbers, extensional databases, therefore side effects are necessary to implement these protocols completely and transparently. For example, to avoid unnecessary communication with a DNS server, the variant of `lookup` in Figure 4 maintains a cache of hostname-address bindings. Before contacting a name server for its address records, the local cache is checked (Line 4). If no cache exists for the given hostname, other name servers are contacted (Line 7) and if the result is non-empty, the local cache is updated (Lines 12–14). The rest of the algorithm remains unchanged.

```

1. declare function Server:multicast($msg) {
2. { local:deliver($msg);
3. let $soa := Server:resources()/soa/@dom return
4. if ($soa) then {
5. for $ns in Server:resources()/ns,
6. $a in Server:resources()/a
7. where $ns/@serv=$a/@host
8. and U:hostname-lt($soa,$ns/@dom)
9. return
10. let server S implement Server at $a/@address return
11. at server S do S:multicast($msg)
12. } else ()
13. }
14. };

```

Figure 5: Multicast, using asynchrony

Caching policies could be transparent to the user, i.e., the DXQ compiler might be able to derive a caching variant of the DNS resolver automatically. In DRM applications, however, caching policies can be quite complex and depend on, for example, the information provider and other external parameters, so the ability to implement such policies explicitly in the same framework is powerful.

3.2 Asynchrony

Another feature of DRM protocols is asynchronous, periodic communication. In the example of Narada, each server periodically sends asynchronous messages to their neighbors reporting their current “view” of the overlay mesh. Multicast services also depend on asynchronous communication. Figure 5 contains the DXQ code for multicast built on top of DNS. Upon receiving a multicast message, a server calls its own server-specific delivery function (Line 2). If the server is the start of authority for some domain (Lines 3–4), it computes the set of nameservers that are contained within its domain (Lines 5–8), then sends an asynchronous multicast message (Line 11) to each such nameserver. This example illustrates the simplicity of implementing a distributed service in DXQ.

Many protocols have several periodic events that run simultaneously. For example, to implement the Narada protocol, the main server function makes one asynchronous call to itself to start each periodic function. The following code snippet illustrates this idiom.

```

let server Self implement Narada at U:self()
return {
  at server Self do ...periodic function 1...;
  at server Self do ...periodic function 2...;
}

```

The utility function `self()` returns the URI of the local explicitly in the same framework is powerfulserver. When a

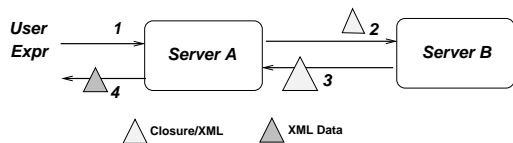


Figure 6: DXQ Servers

program sends an asynchronous message to its own server, a new thread is created in which the body of the `at-server-do` expression is evaluated.

4. DXQ IMPLEMENTATION

A DXQ server exports a module of DXQ functions that may be called remotely by a client application or by another DXQ server. In addition, a DXQ server can evaluate *any* DXQ expression submitted by a requester, which may include calls to the server’s exported functions.

Figure 6 depicts the interaction of two DXQ servers, which can be generalized to any number. Server A accepts a DXQ expression from a user application (Step 1), compiles the expression given the server’s module context, evaluates the resulting expression, and returns an XML value (Step 4). To compute the result may require that some fragment of the compiled expression be evaluated by Server B. In this case, Server A constructs a *closure* [2], which encapsulates the expression with the local context, e.g., dynamic data and function definitions, that is necessary to evaluate the expression remotely. The closure is serialized in XML and shipped to Server B (Step 2).

In response to Server A’s request, Server B server evaluates the expression and, most commonly, returns an *extensional* XML value. In addition, Server B may return an *intensional* expression [14] that the requester may evaluate to compute the result itself (Step 3). For example, assume that Server B implements the module in Figure 2 and it receives the following query from Server A:

```
Server:resources()/a[@host="192.20.3.54"]
```

The extensional result is:

```
<a host="www.research.att.com" addr="192.20.3.54"/>
```

The following expression corresponds to a valid intensional result, which includes Server B’s resource records followed by the path expression to select the host:

```
<rr>
  <soa dom="research.att.com"
    serv="ns.research.att.com"/>
  <a host="www.research.att.com" addr="192.20.3.54"/>
  <ns dom="." serv="a.root-servers.net"/>
  <a host="a.root-servers.net" addr="198.41.0.4"/>
</rr>/a[@host="192.20.3.54"]
```

Intensional answers have (at least) two uses. First, they may require less work to compute than an equivalent extensional answer, and so they permit a server to shift work to the client, for example if the server is overloaded. Second, they can be used to collapse a distributed recursive program to a single site. Consider a function `A()` exported by Server A, which calls a function `B()` exported by Server B, which in turn calls `A()` at Server A. If Server B returns an intensional answer to Server A instead of invoking `A()`, the loop will be collapsed to Server A, which can then evaluate it more efficiently.

```
6. from server S return S:resources()/a[@host=$n],
7. for $a1 in from server S return {
8.   for $ns in S:resources()/ns,
9.     $a2 in S:resources()/a,
10.    where $ns/@serv=$a/@host
11.    and fn:not($ns/@dom = ".")
12.    and U:hostname-1t($ns/@dom,$n)
13.    return $a2
14. }
15. return Resolver:lookup($a1/@addr, $n)/a
```

Figure 7: Alternative resolver

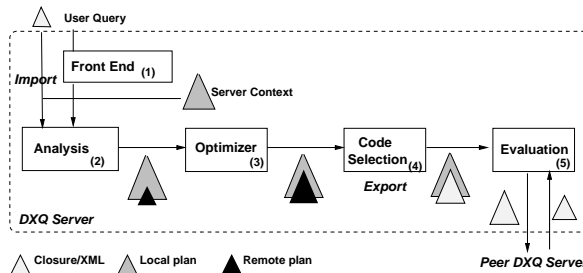


Figure 8: DXQ Server Architecture

4.1 Distributed Optimization

DXQ’s ability to move arbitrary expressions between servers enables some interesting distributed query optimizations. In a network setting, the most important concerns are to reduce the number of messages exchanged by servers and to reduce the amount of data moved between servers.

For example, if the query in Figure 3 is evaluated as written, the resolver will ask for and receive all of a name server’s resource records, even though most are irrelevant to looking up a specific domain name. The alternate query of Figure 7 indicates a more efficient plan for computing the same result: A remote server is contacted twice, once to return just the address records for the hostname (Line 6), and once to return the addresses of name servers that can resolve the hostname (Lines 7–14). As in Figure 3, Lines 11–12 select nameservers whose domains contain the hostname `$n`. In this variant, the server `S` does some of the work that was done by the resolver in the first algorithm (selecting the address records for the hostname and computing the addresses of name servers that can resolve the hostname).

A key technical problem in the implementation of DXQ is to rewrite an expression, whether a user’s query or a remote closure, into an extensionally equivalent expression that is more efficient. Optimizing DXQ is challenging: optimization techniques for XQuery are still nascent, and optimization of distributed queries is a notoriously hard problem. Before describing the heuristic optimizations currently implemented, we describe the architecture in which DXQ is implemented.

Each DXQ server contains a complete, threaded Galax XQuery processor. Figure 8 depicts the engine’s processor. Upon receiving an expression to evaluate, i.e., a user query, a remote closure, or an expression whose target is the server itself, the server creates a new thread in which to evaluate the expression. The front end compiles the module exported by the server and user’s queries into algebraic query plans [7, 20], which is the program representation exchanged by DXQ servers in a closure. When a server receives a user query (or a remote closure), it compiles the query into an algebraic

plan (or respectively, extracts the algebraic plan from the remote closure), then analyzes the plan with respect to the server’s module context (Steps 1-2). The resulting plan may contain a fragment that will be evaluated remotely. The optimization phase applies heuristic rules to rewrite the plan, possibly resulting in a plan in which a larger fragment is evaluated remotely (Step 3). Code selection selects physical operators for each algebraic operator in a plan; for the remote-evaluation operator, the remote plan is exported into XML as the code fragment of the closure (Step 4). Lastly, the physical plan is evaluated. When a remote-evaluation operator is evaluated, the data fragment of the closure is computed (Step 5), and the complete closure is shipped to a peer server.

Our current optimizer pushes selections through the remote-evaluation operator, including value predicates and path expressions; pushes joins through a remote evaluation when the data for both branches of the join are at the same location; and merges sequential remote-evaluation expressions to the same location. These simple rules permit the query in Figure 3 to be transformed into the query in Figure 7. More sophisticated techniques are the subject of current and future work.

5. RELATED RESEARCH

Systems like Astrolabe [23] demonstrate that adding programmability to a hierarchical database like DNS greatly increases its utility for DRM applications. Astrolabe creates a hierarchy of *zones*, each of which exports a database that can be accessed using SQL. Furthermore, a zone’s database can be defined by an aggregate SQL query over the databases of child zones, and database contents are updated automatically as state changes. Services like multicast are built on top of this infrastructure by adding a separate agent in each zone that consults the local database and communicates with agents in other zones. We have an integrated approach that allows us to write resolvers and services like multicast entirely within DXQ. This has two potential advantages over systems that simply expose a database API as a library. First, services are easier to write because DXQ is a high-level language and database optimizations can be applied automatically *across* distributed servers. Second, it is not necessary to pre-deploy agents in the network; for example, in DXQ, multicast can be written at the client, and the necessary code for implementing the multicast migrates as necessary through the network. This allows for easy experimentation, while still allowing code to be pre-deployed for efficiency if desired.

There are several other DRM systems based on database languages. DXQ’s remote execution operator was inspired by Jim’s previous work on using distributed databases for security policies [10, 12] and SD3 [11]. Like SD3 and d3log [12], DXQ supports both extensional and intensional answers to distributed queries. Intensional answers permit DXQ to detect and recover from (potentially infinite) recursion between multiple servers.

Sophia [24] is an “information plane” for networked systems. Its query language is a logic programming language that extends Prolog. Like DXQ, Sophia provides an explicit remote evaluation expression, whose target may be computed dynamically; loadable server modules; and distributed query optimization via query shipping. Sophia includes continuous updates, but lacks asynchronous execu-

tion and XML support. NDlog [13] and SeNDlog [1] are distributed variants of Datalog that have been used for monitoring network properties like connectivity and for implementing distributed security policies. Even though the motivation for these is similar to that of DXQ, the use of a rule-based, logic languages limits their applicability to large scale applications. Implementation of many features necessary in distributed protocols is convoluted: updating counters and protocol state, and APIs to host languages are cumbersome, and the absence of modularity makes programs exceeding fifty or more rules hard to understand and modify. Our goal is for DXQ to hit a “sweet spot” between general-purpose programming languages and telegraphic rule-based languages.

XL [8] was the first XML-based scripting language designed to implement Web services. XL inspired many of the features in XQueryP, such as XML updates and invoking the operations of remote Web services. Moreover, XL has explicit primitives for parallelism. DXQ expands upon these features by shipping arbitrary query plans to remote servers, generalizing the invocation of WSDL operations in XL and XButler [15]. XQueryD [19] supports remote execution of XQuery via an expression like DXQ’s `from-server`, and adds exception handling, but lacks DXQ’s updates, asynchronous evaluation, and the ability to declare interfaces for servers and dynamically bind implementations to interfaces. XRPC [26] adds distributed queries and distributed updates to XQuery, and provides a “loop-lifting” optimization for `for` expressions whose bodies make remote function calls. The `for`-loop bodies are shipped remotely in a bulk RPC, but XRPC does not optimize across the function-call itself. Distributed XML-Query [22] is a protocol for shipping XQuery expressions and XML results between XQuery servers, but provides none of the distributed linguistic features or optimization capabilities of DXQ.

6. DISCUSSION

We are pleasantly surprised by the simplicity, utility, and expressive power of DXQ, are excited by its potential applications, and challenged by the problems to solve before DXQ can be used by other XQuery users. We discuss several problems and opportunities.

Writing and running DXQ programs is fun, but debugging can be exasperating. To help us understand DXQ applications and to better explain DXQ’s functionality, we have developed a GUI. DXQ servers report their existence and summaries of messages to a central GUI server, which visualizes the inter-server message traffic. Inter-server message traffic helps us understand “who is talking to who about what”, but doesn’t help us understand local state, that is why we expose our graph-drawing API in DXQ so that applications can explicitly report state change. We find that the simultaneous visualization of “fast” message traffic with “slow” state changes helps us better understand and debug DXQ programs.

DXQ is a concurrent programming language: The evaluation of each synchronous or asynchronous remote expression occurs in a separate thread. Concurrent access to shared values, e.g., global variables in a server’s exported module, must be protected by locks. DXQ provides functions for creating, locking, and unlocking mutexes, but the DXQ user is responsible for applying the mutexes correctly. This solution permits us to prototype applications, but is untenable

in the long run as simple errors lead to deadlocks and race conditions. Our next task is to investigate synchronization models for DXQ.

One potential use of DXQ is as an implementation language for Web services. In previous work [15], we described how to expose an XQuery module as a Web service and to import Web services, described in WSDL, into XQuery programs. Inter-service communication, however, was restricted to synchronous function calls. With DXQ, we can support both synchronous and asynchronous messaging between Web services. More significantly, DXQ's programming model can generalize the Web-service interface, making it more "transparent" by permitting evaluation of arbitrary expressions in addition to method calls. This capability could help address the performance problems of Web-service implementations.

The interactions between a server's evaluation policies and the ability to ship arbitrary expressions and receive intentional answers are subtle and potentially conflicting. For example, one DXQ server may optimize a query, which results in shipping a "join" operator to another server, which may decide that the "join" operator is too expensive and return an intensional result.

One way in DXQ to account for a server's evaluation policies or capabilities is to incorporate them into the cost model of a cost-based optimizer. For example, a server that does not permit evaluation of recursive functions shipped from another server would export a cost model in which recursive functions have an infinite cost. Similar techniques can be used to prohibit clients from shipping "expensive" operators, like joins. We look forward to working on these challenging problems.

7. REFERENCES

- [1] M. Abadi and B. T. Loo. Towards a declarative language and system for secure networking. In *NetDB '07: Proceedings of the 3rd International Workshop on Networking meets Databases*, 2007.
- [2] A. W. Appel. *Modern Compiler Implementation in C/Java/ML*. Cambridge University Press, 1998.
- [3] D. Chamberlin, M. Carey, D. Florescu, D. Kossmann, and J. Robie. XQueryP: Programming with XQuery. In *XIME-P 2006*, Chicago, IL, USA, June 2006.
- [4] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), 2002.
- [5] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using a peer-to-peer lookup service. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002.
- [6] M. Fernández, T. Jim, K. Morton, N. Onose, , and J. Siméon. Highly distributed XQuery with DXQ. In *Proceedings of ACM Conference on Management of Data (SIGMOD), Demonstration Program.*, June 2007.
- [7] M. Fernández, P. Michels, J. Siméon, and M. Stark. XQuery streaming *à la carte*. In *ICDE*, Istanbul, Turkey, Mar. 2007.
- [8] D. Florescu, A. Grünhagen, and D. Kossmann. XL: a platform for web services. In *CIDR*, 2003.
- [9] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. V. Reich. The Open Grid Services Architecture, version 1.5. Technical Report GFD.80, Global Grid Forum, 2006. <http://www.ggf.org/documents/GFD.80.pdf>.
- [10] C. A. Gunter and T. Jim. Policy-directed certificate retrieval. *Software Practice and Experience*, 30(15):1609–1640, 2000.
- [11] T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106–115, 2001.
- [12] T. Jim and D. Suci. Dynamically distributed query evaluation. In *PODS*, 2001.
- [13] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, pages 97–108, New York, NY, USA, 2006. ACM Press.
- [14] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. D. Ngoc. Exchanging intensional XML data. *ACM Trans. Database Syst.*, 30(1):1–40, 2005.
- [15] N. Onose and J. Siméon. XQuery at your web service. In *Proceedings of International World Wide Web Conference*, pages 603–611, New York, NY, USA, 2004. ACM Press.
- [16] K. Park, V. S. Pai, L. Peterson, and Z. Wang. CoDNS: Improving DNS performance and reliability via cooperative lookups. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.
- [17] L. Peterson and J. Wroclawski. Overview of the GENI architecture. Technical Report Design Document 06-11, Global Environment for Network Innovations, 2006. <http://www.geni.net/GDD/GDD-06-11.pdf>.
- [18] V. Ramasubramanian and E. G. Sizer. The design and implementation of a next generation name service for the Internet. In *Proceedings of SIGCOMM*, Aug. 2004.
- [19] C. Re, J. Brinkley, K. Hinshaw, and D. Suci. Distributed XQuery. In *Workshop on Information Integration on the Web*, pages 116–121, 2004.
- [20] C. Re, J. Simeon, and M. Fernández. A complete and efficient algebraic compiler for XQuery. In *ICDE*, Atlanta, Georgia, Apr. 2006.
- [21] Internet domain survey. <http://www.isc.org/ops/ds/>, July 2006.
- [22] C. Thiemann, M. Schlenker, and T. Severiens. Proposed specification of a distributed XML-query network, 2003.
- [23] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.
- [24] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: an information plane for networked systems. *SIGCOMM Comput. Commun. Rev.*, 34(1):15–20, 2004.
- [25] D. Wessels, M. Fomenkov, N. Brownlee, and K. Claffy. Measurements and laboratory simulations of the upper DNS hierarchy. In *Passive and Active Network Measurement Workshop (PAM)*, Apr. 2004.
- [26] Y. Zhang and P. Boncz. Loop-lifted XQuery RPC with deterministic updates. Draft manuscript, Centrum voor Wiskunde en Informatica, Nov. 2006.

APPENDIX

A. ON-LINE DEMONSTRATION.

All DXQ examples in Section 2 are available in an on-line demonstration at <http://db.ucsd.edu/dxq/>. The demonstration includes a graphical depiction of an example DNS network. Manual playback of each example query illustrates the communication patterns, with and without query optimization, and displays the closures and XML values contained in each message.